OCR Computer Science A Level

2.1.3 Thinking Procedurally

Advanced Notes

<u>**Specification:**</u>

2.1.3 a)

- **Identify the components of a problem**

2.1.3 b)

- **Identify the components of a solution to a problem**

2.1.3 c)

- **Determine the order of the steps needed to solve a problem**

2.1.3 d)

- **Identify sub-procedures necessary to solve a problem**

Identify the components of a problem

In computer science, thinking procedurally makes the task of writing a program a lot simpler by breaking a problem down into smaller parts which are easier to understand and consequently, easier to design.

The first stage of thinking procedurally in software development involves taking the problem defined by the user and breaking it down into its component parts, in a process called problem decomposition. In this process, a large, complex problem is continually broken down into smaller subproblems which can be solved more easily. By separating the problem into sections, it becomes more feasible to manage and can be divided between a group of people according to the skill sets of different individuals.

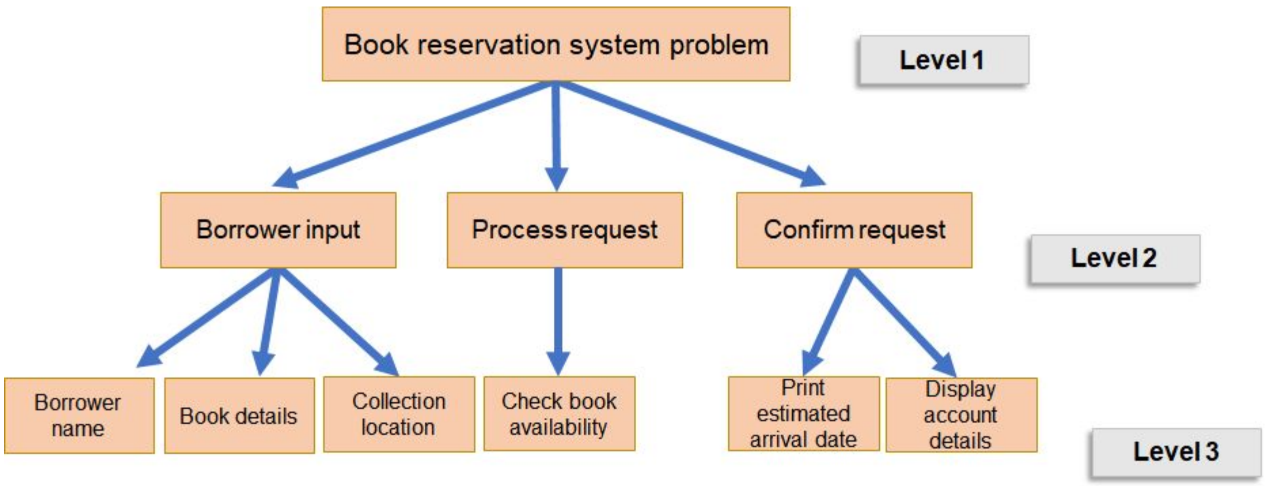
This process requires software developers to consider a problem in terms of the underlying subproblems that need to be solved to achieve the desired result. An adventure game, for example, might be broken down into the following components:

1. Characters
2. Adventures
3. Enemies

These would then be broken down further, as shown:

1. Characters
   - Characters' interactions
   - Characters' appearance
2. Adventures
   - Levels
   - Backgrounds and settings
3. Enemies
   - Enemies' interactions
   - Enemies' appearance

Problems are commonly decomposed using top-down design, shown below.



This is also known as stepwise refinement, and is the preferred method used to approach very large problems, as it breaks problems down into levels. Higher levels provide an overview of a problem, while lower levels specify in detail the components of this problem.

Synoptic Link

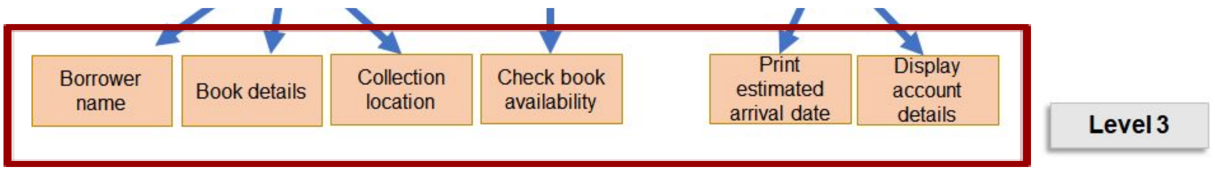
You will have encountered the idea of breaking a problem down into **levels of complexity** in **2.1.1, Thinking Abstractly**.

The aim of using top-down design is to keep splitting problems into subproblems until each subproblem can be represented as a single task and ideally a self-contained module or subroutine. Each task can then be solved and developed as a subroutine by a different person. Once programmed, subroutines can also be tested separately, before being brought together and finally integrated.

Identify the components of a solution

This is the stage in which the details about how each component is implemented are considered. You will be able to see below how separating out these components has made it easier to identify a feasible and programmable solution.

In the same way that we broke down the problem, we must also build up to its solution. In order to identify the components of the solution, each programmer must evaluate the component of the problem allocated to them and assess how it can best be solved. Going back to our previous example involving the book reservation system, we need to consider the lowest-level components.



<u>Borrower name</u>
This could be implemented as a procedure, `getName()`, which checks to see whether or not a user is signed-in to their library account. If they are already signed-in, their name can be retried by querying the library's database of users for the name of the borrower associated with the borrower's ID. Users that are not signed-in should be redirected to a page, requesting them to either register or sign-in. These options should redirect the user to the relevant form.

Synoptic Link

SQL is a language commonly used to **query databases**. You will have come across SQL in **1.3.2**.

The query described here might look like this:

```
SELECT borrower_name
FROM borrowers_table
WHERE borrower_id='392';
```

Book details
The user should be able to enter the name of the book into a text entry field, which would display the books stocked by the group of libraries. This could be implemented as a function which returns the ISBN of the selected book, which is easier to handle and can be more useful than a string.

Collection location
This input could also be implemented as a function, which returns the location specified by the user. It is impractical to use a text entry field here, as this raises the likelihood of erroneous data being entered, such as a location where a library does not exist. Therefore, this data is best collected through a drop-down field in a form.

Checkbook availability
Another database query would have to be carried out to check whether books under the selected ISBN are currently on loan or available for borrowing. This problem could be programmed as a function which returns 'True' if the book is available, or 'False' if not.

As an exercise, consider the ways in which the two final modules could be implemented.

During this stage, it is also useful to identify tasks which could be solved using an already existing module, subroutine or library. Reducing the complexity of the development stage by picking up on sections in which reusable components can be used is another benefit of using top-down design.

Synoptic Link

There are **many advantages** to **reusing existing components**. These are discussed in **2.1.2**.

Finally, the components of the solution are combined to form a full, working solution.,

Order of steps needed to solve a problem

When constructing the final solution based on the solutions to the problem components, thinking about the order in which operations are performed becomes important. Some programs might require certain inputs to be entered by the user before the processing can be carried out. These inputs would also need to be validated before they can be passed onto the next subroutines, which must also be taken into consideration.

It might be possible for several subroutines to be executed simultaneously within a program, and programmers must identify where this is possible by looking at the data and inputs the subroutine requires. Some subroutines will require data from other subroutines before they are able to execute, and so will be unable to execute simultaneously. In this case, programmers should determine the order in which subroutines are executed, as well as how they interact with each other, based on their role in solving the problem.

Going back to our earlier example involving a book reservation system, it is clear that certain operations must be performed before others can be performed. Without the user's input, the rest of the program cannot execute. There must also be checks in place to confirm the validity of these inputs before data is passed between subroutines.

The same principle is important when considering how a program will be used. Programs should be built so as to ensure operations are not carried out in an order that does not make sense, or will raise an error. Consider an adventure game. It should not be possible for users to access and play levels ahead of those they have unlocked. A fast food delivery app should not allow users to select food until they have confirmed their location, nor should it allow users to pay before they have confirmed their order. Although these things may seem obvious and natural to us, they must be explicitly written into software for programs to work in the way we want.